第二章 线性表

一、线性表的逻辑结构

1. 线性表的定义：①线性表是n个系统数据元素的有限序列

②线性表示一种数据结构/逻辑结构.

2. 线性表的特点：①表中元素的个数有限

②表中元素具有逻辑上的顺序性

③表中元素的数据类型都相同，这意味着每个元素占有相同大小的存储空间.

二、线性表的运算/操作

InitList(&L)	初始化表
Length(L)	求表长
LocateElem(L,e)	按值查找
GetElem(L,i)	按位查找
ListInsert(&L,i,&e)	插入
ListDelete(&L,i,&e)	删除
PrintList(L)	输出
DestroyList(&L)	销毁

三、线性表的顺序表示.

1. 顺序表相关概念：

(1)定义：用一组地址连续的存储单元依次存储线性表中的数据元素，从而使得逻辑上相邻的两个元素，在物理位置上也相邻

(2)特点：①表中元素的逻辑顺序与物理顺序相同

②随机访问，即通过首地址和元素序号能在时间O(1)找到指定的元素.

③顺序表的存储密度高，每个节点只存储数据元素.

④顺序表逻辑上相邻的元素，物理上也相邻，所以插入和删除需要移动大量元素.

2. 静态分配和动态分配

	静态分配	动态分配
定义	①数组的大小和空间事先已经固定 ②一旦空间占满，再加入新的数据就会产生溢出，进而导致程序崩溃	①存储数组的空间是在程序执行过程中通过动态存储分配语句分配的 ②一旦空间占满，就另外开辟一块更大的存储空间用以替换原来的存储空间

顺序存储类型	#define MaxSize 50 //最大长度
	typedef struct
	{
	ElemType data[MaxSize]; //顺序表元素
	int length; //当前长度
	}SqList; //顺序表类型定义
		#define InitSize 100 //表长度初始定义
		typedef struct
		{
		ElemType *data; //指示动态分配数组的指针
		int MaxSize, length; //数组的最大容量和当前个数
		}SeqList; //动态分配数组顺序表的类型定义

3. 动态分配语句

C： L.data = (ElemType*) malloc (sizeof(ElemType)*InitSize);

C++： L.data = new ElemType[InitSize];

4. 顺序表相关操作

(1) 插入

```
bool ListInsert(SqList &L, int i, ElemType e)
{
  if(i<1 || i>L.length+1)        // i 范围无效(1~n+1)
    return false;
  if(L.length >= MaxSize)        // 空间已满
    return false;
  for(int j=L.length; j>=i; j--) // 将第i个元素及以后的元素后移
    L.data[j] = L.data[j-1];
  L.data[i-1]=e;                 // 位置i处放入e
  L.length++;
  return true;
}
```

最好情况：O(1)
最坏情况：O(n)
平均情况：O(n)

(2) 删除 bool ListDelete(SqList &L, int i, ElemType e)

```
{
  if(i<1 || i>L.length+1)        // 范围有效
    return false;
  e=L.data[i-1];                 // 将被删除的元素赋值给e
  for(int j=i; j<length; j++)    // 将第i个元素以后的元素前移
    L.data[j-1]= L.data[j];
  L.length--;
  return true;
}
```

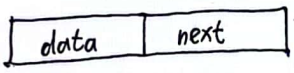
最好：O(1)
最坏：O(n)
平均：O(n)

(3) 按值查找 int LocateElem(SqList &L, int i, ElemType &e)

```
{
  int i;
  for(i=0; i<length; i++)
    if(L.data[i]==e)             // 下标i的元素值为e，返回其位序i+1
      return i+1;
  return 0;
}
```

最好：O(1)
最坏：O(n)
平均：O(n)

四、线性表的链式表示

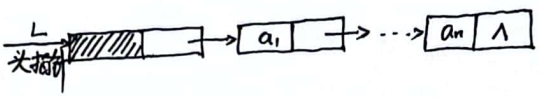


非随机存取的存储结构，查找特定结点，从头遍历。

1. 单链表结点类型描述如下：

```
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}LNode, *LinkList;
```

2. 带头结点的单链表：



① 第一个位置与其他位置的操作一致

② 空表、非空表处理统一。

3. 单链表初始化

```
bool InitList(LinkList &L){
    L=(LNode*)malloc(sizeof(LNode));
    L->next=null;        //创建头结点，之后还没有元素结点
    return true;
}
```

4. 求表长

时间复杂度 O(n)

```
int Length(LinkList L){
    int len=0;
    LNode *p=L;
    while(p->next!=NULL){
        p=p->next;
        len++;
    }
    return len;
}
```

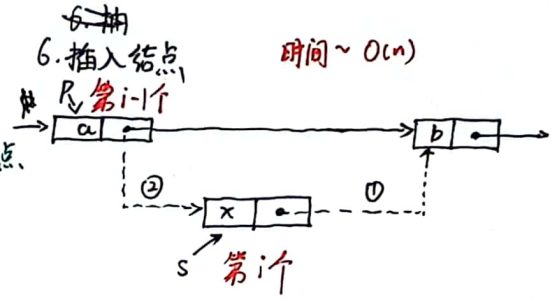
5. 按序号查找结点

时间 ~ O(n)

```
LNode *GetElem(LinkList L, int i){
    LNode *p=L;
    int j=0;
    while(p!=NULL && j<i){
        p=p->next;
        j++;
    }
    return p;
}
```

6. 插入结点

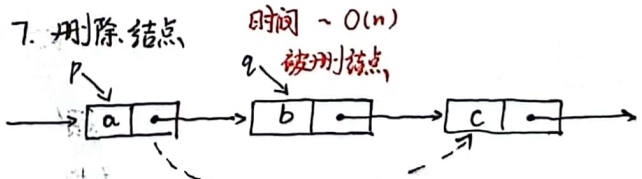
时间 ~ O(n)




```
bool ListInsert(LinkList &L, int i, ElemType e){
    LNode *p=L;
    int j=0;
    while(p!=NULL && j<i-1){
        p=p->next;
        j++;
    }
    if(p==NULL) return false;
    LNode *s=(LNode*)malloc(sizeof(LNode));
    s->data=e;
① s->next=p->next;      若先②，原链接断，b找不到。
② p->next=s;
    return true;
}
```

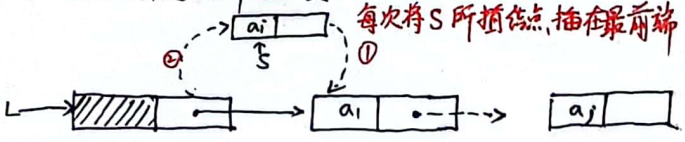
7. 删除结点

时间 ~ O(n)




```
bool ListDelete(LinkList &L, int i, ElemType &e){
    LNode *p=L;
    int j=0;
    while(p!=NULL && j<i-1){
        p=p->next;
        j++;
    }
    if(p==NULL || p->next==NULL) return false;
    e=q->data;
    p->next=q->next;
    free(q);
    return true;
```

8. 头插法建立单链表 时间 O(n)



每次将S所指结点插在最前端

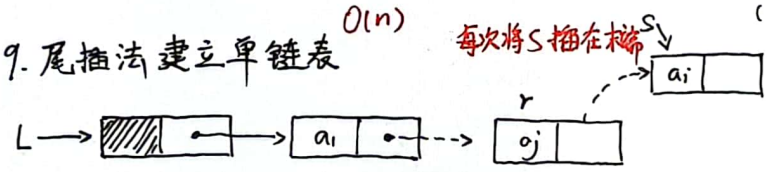
~~LinkList List Head~~

```
LinkList List_HeadInsert (LinkList &L) {
    LNode *S; int x;
    L=(LNode*) malloc (sizeof (LNode));
    L->next = NULL;
    scanf ("%d", &x);
    while (x!=999) {
        S=(LNode*) malloc (sizeof(LNode));
        S->data = x;
        ① S->next = L->next
        ② L->next = S;
        scanf ("%d", &x);
    }
    return L;
}
```

读入数据的顺序与生成的链表中元素的顺序相反.

9. 尾插法建立单链表 O(n)

每次将S插在表尾



增加一个尾指针r, 指向当前链表的尾结点.

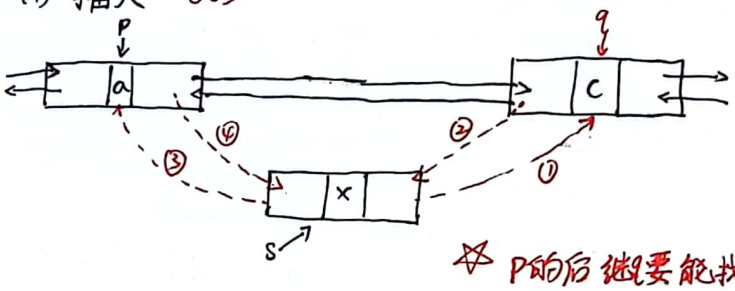
```
LinkList List_TailInsert (LinkList &L){
    int x;
    L=(LNode*) malloc (sizeof(LNode));
    LNode *S, *r=L;
    scanf ("%d", &x);
    while (x!=999) {
        S=(LNode*) malloc (sizeof(LNode));
        S->data = x;
        r->next = S;
        r=S;
        scanf("%d", &x);
    }
    r->next = NULL;
    return L;
}
```

10. 双链表结点类型描述

```
typedef struct DNode{
    ElemType data;
    struct DNode *prior, *next;
}DNode, *DLinkList;
```

插入和删除保证在修改的过程中不断链.

(1) 插入 O(1)

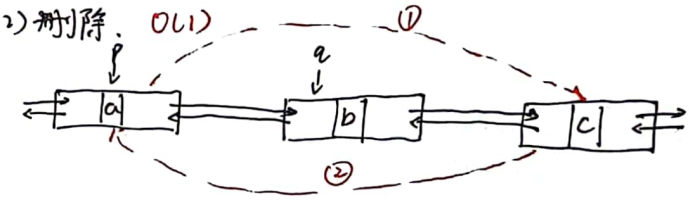


P的后继q要能找到, q的前驱才能指向s

先连后断

① S->next = p->next;
② p->next->prior = S;
③ S->prior = p;
④ p->next = S;

(2) 删除. O(1)



① p->next = q->next;
② q->next->prior = p;
free(q);

11. 循环链表

(1) 循环单链表: 表尾结点 r->next = L, 无NULL指针结点.

① 判空条件: 头结点的指针域与L的值相等.

② 可以从表中任意一个结点开始遍历链表.

③ 设头指针: 表尾插入, O(n)
设尾指针: r->next即为头指针, 表头/尾插入, O(1)

(2) 循环双链表: 头结点的prior指针指向尾结点.

① *p为尾结点时, p->next = L.

② 判空: L->prior == L && L->next == L

12. 静态链表

用数组来描述线性表的链式存储结构，需预先分配一块连续的内存空间。

```
#define MaxSize 50
typedef struct {
    ElemType data;
    int next;    //下一个元素的数组下标
}SLinkList[MaxSize];
```

① 以 next = -1 作为结束的标志。

② 插入和删除，不需移动元素。

五、顺序表和链表的比较

	顺序表	链表
读取方式	能随机存取	不能随机存取
逻辑结构与物理结构	逻辑相邻，物理相邻	逻辑相邻，物理不一定相邻
空间分配	预先按需分配存储空间	在需要时申请分配，只要内存有空间就可
按序号查找	无序O(n)，有序折半O(log₂n)	O(n)
按值查找	O(1)	O(n)
插入	O(n)	O(1)
删除	O(n)	O(1)

六、怎样选取存储结构

	顺序表	链表
基于存储	难以估计规模，不宜采用；存储密度高	不用估计；存储密度低
基于运算	按序访问，还	插入，删除
基于环境	易于实现	基于指针

1. 顺序表中所有"元素的类型必须相同，且必须连续存放，一维数组中元素可以不连续存放。

2. 顺序表所占的存储空间 = 表长 × sizeof(元素类型)。

3. 链式存储设计中，各个不同结点的存储空间可以不连续，但结点内的存储单元地址必须连续。

4. 顺序存储方式可用于存储线性结构、树和图。

5. 在一个没有头指针和尾指针的单链表中，删除表尾结点时，必须从头开始找到表尾结点前驱，O(n)。

6. 队列在表头删除，表尾插，可采用带尾指针的循环链表。

7. 带头结点单链表空：L->next == NULL ；不带头结点空：L == NULL 。

8. 常在末尾插和删结点，选用带头结点的双循环链表，因为需要寻找尾结点及尾结点的前驱结点。

9. 常删除第一个元素，最后一个元素；在第一个元素之前插入，在最后一个元素之后插入，选用只有头结点指针，没有尾结点指针的循环双链表。

10. 常在最后一个元素插入和删除第一个元素，考虑不带头结点且有尾指针的单循环链表。

11. 常删除最后一个元素，最好使用带尾结点的双链表或者带任意结点的循环双链表。